Reinforcement Learning and Blackjack

King's Certificate

Adam Scriven

June 25th, 2021

Rafael Davison, Sahil Gorci, Rebecca Jesson, Dev Thakkar

Content

Acknowledgement

We would like to thank Adam Scriven for teaching us much about Reinforcement Learning, from the major concepts to the minutiae of python programming. A special thanks goes for always making time for our questions inside and outside scheduled meetings, as well as all other computing or life-related advice. We would also like to thank Emma Lawsen for organising and making these projects possible as well as always providing helpful feedback to help us develop our research and collaboration skills. Many thanks to KCLMS for providing us with the resources and time to complete our research in an appropriate timing.

Abstract

Our project has been to explore techniques of reinforcement learning and put its theoretical principles into practice. One way to test this is through the familiar structure and well-defined rules that games provide, with wins and losses as an accurate measure of success. Using a case study of blackjack provided us with a small state space game to use as a sandbox for different techniques, allowing us to explore their benefits and drawbacks from a first-hand practical perspective. Amongst these we tried Monte Carlo (exploring starts) and greedy policy algorithms to play a basic form of blackjack, then we moved on to try the concept of expanding state space through card counting, as well as symmetrical self-play.

1 Introduction

The ideas behind reinforcement learning are an inherent part of human nature and the concept of learning. When a child learns to walk, it does not have an explicit teacher, learning instead as a result of the constant feedback it receives from its environment. This method of learning is similarly displayed in all living creatures, as shown by a memory test performed in 2013 on a chimpanzee. He was shown the numbers 1 to 9, which then disappeared in a randomly generated order. The chimpanzee learnt that remembering and reiterating the sequence resulted in rewards, and soon repeated the sequence in which the numbers had disappeared almost 90% of the time [1]. This study shows that the chimp exhibited behaviour similar to machine learning: his ability to learn and improve from experience was completely coordinated by a result he received each time he found the correct sequence. This a fundamental aspect of reinforcement learning – an agent taking actions in an environment to maximise its cumulative reward.

The usefulness of reinforcement learning is clear, and the field of reinforcement learning is constantly achieving breakthroughs, one of which was the widely publicised success of AlphaZero, created by DeepMind. This game system used reinforcement learning to master a wide variety of games, such as chess, shogi, and even the notoriously difficult Go, and after which it went on to beat not only the world's best human player but also the world-championship program. The amount of training the program needed to master the three games was approximately 9 hours, 12 hours, and 13 hours respectively, in comparison to the thousands of hours it took the humans and programs it beat to optimise their strategies [2]. Using a similar process, Alpha Fold was established a year after the success of AlphaZero. It can accurately predict the shape of proteins, even though they are biologically complex [3]. The uses of this system are boundless and there is enormous potential to help tackle diseases and find new medicines, all of which will be crucial to research and gain a better understanding of these biological processes. The usefulness and power of reinforcement learning is undeniable.

In the past several decades, the field of reinforcement learning has played a crucial role in increasing the effectiveness and efficiency of artificial intelligence. Alongside other machine learning paradigms, it is has been predicted that reinforcement learning will raise the UK GDP by 10.3% before 2030. This will correspond to an additional £232bn – making AI one of the biggest commercial opportunities in today's fast-changing economy [4]. In addition to this, AI is hypothesised to revolutionise many aspects of everyday life, with some examples of this being the improvement of the quality of life with its implementation into the healthcare system, for example creating patient-specific treatments more efficiently, or the stock market using reinforcement learning to maximise profits.

In a similar fashion to DeepMind, we too have a goal of investigating automation; research in automation could reduce the workload on humanity, allowing us to advance in technology at a rapid rate. Like AlphaZero, we decided to focus on a problem dictated

by specific rules, which would provide an environment that allows for the development of unique techniques and the testing of other methods in which we could evaluate their usefulness and assess their efficiency. The popular casino card game, blackjack, has a binary win or loss outcome, and a relatively small state space which makes it similar to any Boolean issues found in the real world and hence makes it extremely compatible with our aims and objectives. The goal of the game is to collect cards with as high a numerical total as possible without exceeding 21 [5]. In 2019, the total gambling market was worth £14.3 billion with the largest share belonging to online gambling. Within this, 6.1% was accounted for by blackjack [6]. We will use this blackjack reinforcement learning problem to explore our project.

However, while there are many known reinforcement methods, it is often hard to find the most effective solution to a problem. Each method has its benefits and disadvantages, while others rely on certain assumptions not always viable. In blackjack, for example, using a small and simple state space would allow us to use the assumption of exploring starts, however this may not allow us to beat the house edge. Implementing techniques such as pseudo-card-counting would lead to a far larger number of state spaces, making brute force methods impossible, and meaning we will have to find another way to find the optimal policy. In uncharted territory such as this, we will have to look into using methods like off-policy exploration.

With the help of our mentor, Adam Scriven, we will research and gain an understanding of reinforcement learning and the algorithms commonly used in this machine learning paradigm. We plan to create a reinforcement learning algorithm that will learn to play blackjack. We will use Python due to its versatility and its position as the most common language used for machine learning. In this paper, we will contribute to the advancement of reinforcement learning by discussing the benefits of using many methods, and how we will implement them to attempt to beat the house edge in blackjack, with blackjack allowing us to show practical implementations of the algorithms we discuss. Our program must balance exploring new actions with exploiting what it already knows to work well. It will have to explore different policies, calculating state values for each, and eventually will find the most effective strategy for blackjack. By the end of this project, we hope to have a clear idea of reinforcement learning, the various methods within it, and a blackjack algorithm created using a variety of methods, that will be able to beat the house edge.

2 Literature Review

As can be seen in the following sources reinforcement learning is a well-documented field with a plethora of papers, books and articles to choose from. Due to the recent surge in popularity of AI since AlphaZero [2], the internet has become flooded with introductory guides and walkthroughs which were a notable aid in early research and understanding of algorithms and concepts [see sources 7 and 8]. Critically these tend to tackle specific case

studies without much of a mathematical framework which made it difficult to translate the knowledge forward to more advanced topics. Due to the rather in-depth mathematical nature of reinforcement learning, more academic papers had significantly less in terms of required prior knowledge but ended up being more complete and useful in the end [see source 5]. Due to this discrepancy, one of the goals of our final article could be to bridge this gap, comprehensively linking the mathematics to simpler examples, without losing utility.

Initially, we explored a multitude of studies with ranging levels of complexity, Dittert's "Monte Carlo Methods in Reinforcement Learning" [7] being simplest. Over this series of articles, Dittert explains the basics of Reinforcement Learning, covering theoretical definitions and more practical implementations of key concepts. In this specific article, Dittert discusses Monte Carlo Methods and how they can be used to estimate the value function and to find optimal policies. He explains the difference between on-policy and off-policy Monte Carlo Methods, as well as finer details of both, initially focusing on the theory of the topic before looking at more practical applications and algorithms. The articles are aimed at an educated adult with no prior knowledge of reinforcement learning, able to understand complex notation and ideas. Dittert has multiple degrees, including a PhD, in Reinforcement Learning. He has also worked researching around this topic, making him a credible and reliable source. This article, as well as the others published by Dittert cover many of the topics relevant to our project, including those we will use to write our program, and so will significantly influence our research. However, Dittert explains many concepts more simply and in less detail than other similar articles, resulting in this not being our main source as, due to its introductory nature, some areas are covered too basically for our research. Because of this, it will be used as the starting point of our research, after which, its usefulness will diminish. Overall, this is a useful, basic source, directed at educated adults, which we will use to help our initial understanding of Reinforcement Learning.

Whilst the article above covers the broad introduction, "Reinforcement learning (RL) 101 with Python" focuses on applying the theory into a practical program. In this article [9], Gerald introduces the realm of reinforcement learning, specifically how it works and what the main thought process is for the main algorithms, supplemented with Python code for further support. The author uses clear presentation and images to present his research in an aesthetic layout such that no reader would be overwhelmed, however, it provides enough detail so even experienced researchers could refresh their memory. This focuses primarily on key starter concepts that will be essential further down the line to build upon. This article will be very useful for us as a team since it covers the main prerequisites to begin our project, including a basic explanation of Monte Carlo algorithms, Python code showing how to implement the learning procedure and develop an environment for the agent. Although this is a well-written article, it lacks the specific details necessary for our project which will be useful later. Hence, we will require more books that can provide us with further analysis. The author is the founder of 'CryptoDatum.io' and has published several popular articles on machine learning, demonstrating his plethora of knowledge and reliability. Furthermore, his article

is a 'summarised article' inspired by the book, "Reinforcement Learning: An Introduction by Andrew Barto and Richard Sutton" - a popular book regarding this sector of AI. This article will act as our foundation alongside the other articles mentioned to base our project to progress in creating our blackjack game with a fully embedded learning environment built upon our understanding of known reinforcement learning algorithms and equations.

Due to the practical application being covered in the previous article, this next source centres on the mathematical elements of our research. As part of the Deep Reinforcement Learning Explained series [8], Torres seeks to illustrate the core principles of this field to beginners concisely. Partnered with the UPG Barcelona Tech and Barcelona Supercomputing Centre, and with experience as a professor who has published over 200 articles, he has the resources and knowledge to teach this kind of course. By beginning with a description of a Frozen Lake example environment for an agent to act in, the article gradually introduces the concepts of a Bellman equation in a few different forms, specifically focusing on finding the values of different states the agent could be in. Step-by-step walkthroughs of the outcomes of set policies give a clear understanding of how the calculations can be programmed whilst tree diagrams help visualization. Critically this article only discusses possible calculations for when the probabilities of different random outcomes in an environment are known to the agent. This hampers the utility of the source for our project since we will need to focus on finding these unknown variables whereas this assumes, we know them. Overall, it becomes clear that this guide is a useful walkthrough of algorithms that has helped us to understand the introductory elements of RL as it is aimed at beginners, however we will struggle to find extensive use for it in the future due to the fact that it can only solve Markov Decision Processes where probabilities are already known, unlike those in a game of blackjack.

This paper also has a specific focal point, however concerning the important Monte Carlo algorithms. The "Reinforcement Learning in Continuous Action Spaces through Sequential Monte Carlo Methods" paper [10] provides a more focused look into how the novel actor-critic approach may resolve an issue of reinforcement learning (RL) algorithms when applied to continuous action spaces, where a faster method of the identification of the highest-valued action is needed. The paper explores the key ways that the actor-critic methods outweigh the current approach; it represents a policy with a memory structure independent of the one used by the value function, hence more specialised and efficient. Andrea Bonarini has a PhD in Computer Engineering and a Masters in Neuro-Linguistics Programming, he has been teaching for around thirty years at the University of Milan and co-found a company called Nova Labs which revolves around robotics. This shows he is highly knowledgeable about the respective field. Each of his associates has an equal amount of experience and the same tenacity for the field as they have each contributed to at least fifty papers and so it makes this paper highly credible as each author is extremely experienced and the paper would not be perceived as biased as the paper consists of statistics and facts to back up the points. Their paper is aimed at university students, who may have some knowledge regarding mathematics and reinforcement learning in general, but a somewhat stronger understanding of Monte

Carlo Simulations and Reinforcement Learning. This paper explains many segments of the Bellman's Equation and further explores the uses of their research in an experiment involving a boat problem where the boat is trying to move from the left bank to the right with a strong non-linear current to the highest of accuracies. This paper contained many of the topics that our group wanted to research and therefore, was significant and useful. However, it was not our main resource [5], largely because the information provided required a stronger understanding of the RL field, and the experiment was not like our blackjack game, which has a binary system, where '0' represents loss and '1' represents winning [10], not how far away from the right bank they are, which is what was calculated in the paper. Its focus was not comparable to the needs of our project, which is its main limitation - hence, its use has been mainly to gain a stronger understanding of the actor-critic approach, which we could change to better fit our uses.

This final source, a book, is the most in-depth, covering the essential algorithms clear and concisely. In the book "Reinforcement Learning – An Introduction" [5], Sutton and Barto seek to describe the methods and algorithms of reinforcement learning with a mathematical framework that can be applied to a wide variety of problems. Created as a university textbook, it is directed at those with a high level of mathematical knowledge who want to rigorously explore the subtle details of different reinforcement learning methods. As a group, we read the first five chapters which focused on the relevant introductory topics that allowed us to build up knowledge about policy iteration, value iteration and Monte Carlo simulation. Although some of the more mathematical proofs were challenging to understand since they are aimed at degree students, the repeated use of example programs with plenty of pseudocode and diagrams demonstrated the approaches and algorithms used comprehensively. The gradual way in which subsequent methods were introduced showed the reader the strengths and weaknesses of each method and has been very useful in helping us plan our final program approach. One limiting factor is the sheer mass of information discussing detailed topics that requires some prior knowledge to filter through, we found that the accessibility of this book truly opened up once we already had a base understanding of what topics would be most useful to us. Sutton and Barto are experienced authors who, between them, have worked at DeepMind and the University of Massachusetts and their collective knowledge allows them to discuss everything from the historical angle to the most cutting edge algorithms that have given us a very broad understanding. In conclusion, Sutton and Barto provide us with plentiful information which will be exceptionally beneficial in our project especially in contrast to the paper by Alessandro Lazaric et al [10] which focuses on a different type of problem.

As touched upon in the introduction, the literature surrounding reinforcement learning can broadly be separated into two categories: the less detailed, scientific guides aimed at those teaching themselves the more basic concepts, and the more analytical academic papers discussing the mathematical framework for more complex problems. Ultimately, the latter equipped us with far more tools for tackling certain problems, especially when focused on

more abstract applications because instead of focusing on one goal or technique the textbook structure [of source 5 specifically] prioritised describing the advantages and disadvantages of different techniques letting the reader choose which would be the best for their own uses. Finally, all these key tools we have gained will allow us to be able to create an efficient, powerful program of blackjack that over time learns the rules and can win against the house advantage from any given state.

3 Methodology and Development

3.1 Introduction

Following further research on reinforcement learning, we have arrived at a deeper understanding of concepts that allows our practical methods to be conducted more efficiently. Throughout the project, a timetable was implemented with the intent of increasing group coordination and improving team working capabilities. As part of our practical programming approach, we implemented RL into a blackjack environment (specifically Monte Carlo Simulation) to increase the win rate, and this solution provided us with several advantages. Firstly, results would show not only what would happen but also the likeliness of each outcome. Due to the amount of data the simulation creates, it becomes easier to create graphs of different outcomes and their chances of occurrence [similar to those found in 5] making it possible to achieve more visual representation. To summarise, this allowed us to gain a more comprehensive understanding of what the data represents and plan our next steps more progressively.

3.2 Time Frame and Organisation

Towards the beginning of our project, we began by establishing deadlines and goals which we planned to achieve throughout the period of our project as can be seen in the Gantt chart (see Figure 3.1) below. The chart allowed us to segment our time more effectively to complete each section of work in the given time.

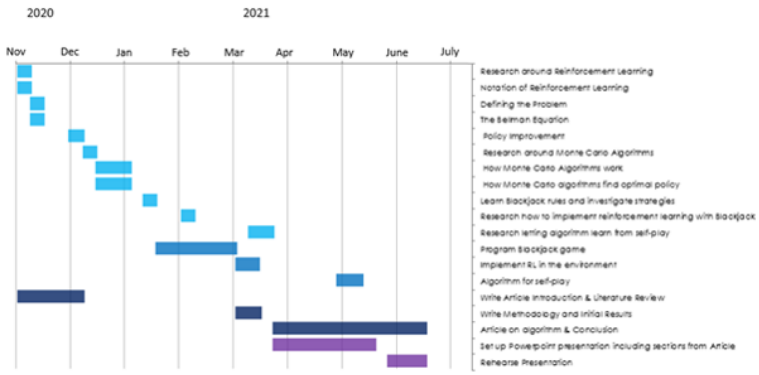

Figure 3.1: Gantt Chart

As shown in Figure 1, we separated our work into four main sections as displayed using different colours. Firstly, we started with our research (as shown in light blue) which can be subdivided into primary and secondary research. Our primary research involved asking questions to our mentor in order to get an experienced view of our topics, meanwhile the secondary research involved browsing through well-known theoretical reinforcement learning literature or otherwise, resources primarily found online that tackled more practical aspects of a programming approach. We used this initial piece of research to produce an introduction and literature review (as shown in the dark blue). Then we began specialising our research so it would be targeted towards the blackjack game to gain a stronger understanding of more useful exploring techniques, explaining this within our methodology. Using all the knowledge we had accumulated, we created an environment for our blackjack game and began to incorporate practical methods (as seen in blue) in which we were able to produce our initial results, along with realistic conclusions to said results. Finally, by compiling all that we had learnt and uncovered using our program, we obtained final results, which would be placed in our article and presentation, along with all other key pieces of data that we had found leading up to this moment.

3.3 Roles and Responsibilities

The roles and responsibilities were shared evenly among the group, with every member working on all parts of the project. In terms of programming, Rebecca wrote the base environment and the Agent-N algorithms, Rafa and Rebecca wrote the initial exploring starts program, and Dev wrote a second version of the program using classes and a greedy policy. From the data collected by these, Sahil and Rafa implemented graphing software to created 3D plots, and other graphs, Sahil from Dev's work and Rafa from the team's.

The card-counting was attempted by Rafa and Sahil, and Rebecca programmed the self-play agents and created heatmaps to show the policies from the exploring starts as well as the self-play, as well as some other diagrams.

This meant that the group was commonly split into two pairs, which resulted in each of the two teams continuously having to maintain communications for both sides to have a strong understanding of each other's progress throughout the project which was held every fortnight through the use of the collaborative app, Teams, or in real life when possible.

The writing was similarly split, with Dev and Sahil working more on parts such as the introduction and the literature review, as well as the methodology, while Rafa and Rebecca had a larger focus on more technical writing, such as that in the sections on results and the key concepts. Decision-making was simplified as whenever an issue arose, we each displayed our views and solutions and effectively concluded to a resolution. The delegation of tasks was controlled by the deadlines for our article and presentation and thus we followed quite closely to a pre-set plan. The tasks would vary on a weekly basis where some would do extensive research on one hand, whilst others may be implementing new methods into our blackjack environment; each task being done by those who felt they were the best suited

for the job and would be able to apply their skills the most effectively. Furthermore, despite the challenges that arise when programming collaboratively, we used tools like repl.it (an online python editor) in order to allow for a smoother workflow and enable teamworking more effectively.

3.4 Conclusion

Our timeframe allowed us to ensure we were always on target, timewise. Cohesive group strategies allowed us to approach learning about our topic with confidence and collaboration, the results of which can be seen below.

4 Key Concepts

4.1 Introduction

Understanding reinforcement learning (RL) is fundamentally about understanding useful exploring techniques which are both efficient and specialised for the job they are intended to do. On the simplest level, RL revolves around the nature of learning, using information that is provided to you to improve. In a more computational approach, RL is goal-orientated and consists of an agent in a specified environment with predetermined rules where it is simply programmed to achieve the highest reward possible. RL problems can be formalised using ideas from dynamic systems theory [9], or in the case of blackjack, through the use of the Markov decision process (MDP). To help solve an MDP, the bellman equation can be used as it is one of the foundation blocks which are omnipresent in RL. Beyond the agent and environment, policies are one of the four main sub-elements of an RL system. Policies define how an agent behaves at a given time and are vital to help simplify the solution [5].

4.2 Markov Decision Processes

Finite Markov decision processes (MDP) are the formal problems that reinforcement learning can be used to solve. They are a formalised framework for sequential decision-making, where actions influence all immediate rewards, future states, and the future rewards caused by those states. As MDPs act in this way, there is a decision that must be made between immediate and future reward.

The learner and decision-maker in an MDP is called the agent, and it will interact with the environment to achieve rewards, which tell the agent how well they performed. The agent and the environment continually interact, with the agent performing actions and the environment responding accordingly and presenting new situations to the agent. Over time, the agent seeks to maximise overall rewards. The process is shown in Figure 4.1. [5]

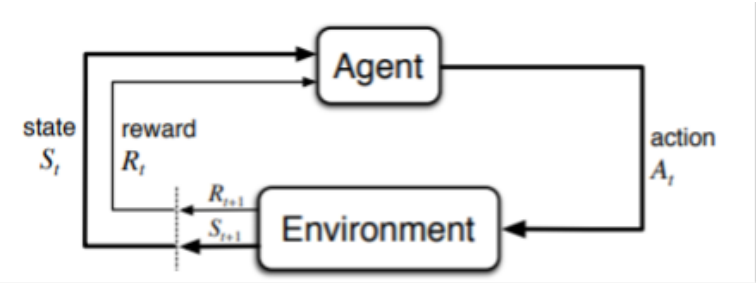

Figure 4.1: The interaction of the agent and environment in an MDP. [5]

As can be seen above, the agent and environment interact at discrete timesteps, shown above as t. At each time step, the agent receives a representation of the environment, called the state, and based on this will select an action. One time step later, the agent will receive the reward that was the consequence of its action and a new state. This process will repeat until the MDP. In a finite MDP, there are a limited number of states, as well as a limited number of actions in each state, and rewards that can be achieved. This results in being able to calculate probabilities for each reward or state from any position. However, these are sometimes too difficult to calculate, meaning that the agent will work without knowledge of their environment.

4.3 Goals and Rewards

In reinforcement learning, the main goal of an agent is to maximise long term reward. [9] Therefore, it must concentrate not on immediate reward, but on which actions will lead to the most reward in the future. Formally, this is the reward hypothesis, and is stated as: “That all of what we mean by goals and purposes can be well thought of as the maximization of the expected value of the cumulative sum of a received scaled signal.” [5]

It is important to ensure that reward teaches the agent the correct idea. For example, if someone was to reward an agent learning chess for taking pieces and occupying certain positions rather than winning, the agent may learn to take pieces at the expense of the game.

4.4 Returns and Episodes

The goal of an agent is to maximise long term reward, and this can be said more generally as maximising the returns of an episode. The return is defined as the sum of the rewards at each timestep. This episodic approach is best for problems where it can be broken up easily into sections, such as for plays of a game or similar. In such a problem, each episode will end in a terminal state before being reset to the starting state. Each episode is independent of previous ones. This means, for example, in a game, previous episodes being lost will not affect the next game. In some episodes, there are many rewards per game, and so to push an agent towards maximising future rewards, discounting is used. This is where each reward is

multiplied by a discount rate as many times as time steps ago it happened. This will produce a discounted return.

4.5 Policies

Agents follow 'policies' these are functions that map the current state, to an action to be taken in that scenario. Reinforcement learning is about finding the policy that eventually will give the highest possible rewards.

For finite MDPs, there is always at least one optimal policy for any problem. In this policy, its expected return is greater than or equal to the expected return from any other policy in all states.

4.6 Monte Carlo Methods

Monte Carlo (MC) methods are learning methods for estimating value functions and discovering optimal policies, solving reinforcement learning problems. [7] They, importantly, do not require any knowledge of the environment, only experience. Yet, even with this limited knowledge, they can still achieve optimal behaviour.

MC methods work by averaging sample returns. Following each episode, value estimates and policies can be updated following the information gained. They will sample and average returns for each state-action pair, creating a policy of the best action to perform in any situation. [7]

As previously mentioned, the value of a state-action pair is the expected return starting from that state. To then estimate this number, we can visit that run our problem many times and record the return following every visit to that state. As the number of times that the state is visited goes to infinity, the estimated value of the state-action pair will approach its true value, and the optimal policy can be found.

4.7 Exploring Starts

If a learning agent simply learns a policy through experience and plays only off that policy, then many state-action pairs may never be visited. This is because, as soon as an agent finds one good action from a state, it will always follow that one, meaning certain routes are never taken. Because of this, the policy will only observe returns for one of the actions from each state. With no returns to average, the MC estimates of the other actions will not improve. This could lead to the agent learning the wrong policy. For MC to work, we need to estimate the values of all the actions from each state, not just the one we currently believe to be the best.

To solve this problem and maintain exploration, the assumption of exploring starts can be used. We must keep continual exploration. One way of doing this is placing the agent in a random state-action pair, where every pair has a probability of occurrence that is greater than

0. By doing this, we can ensure that every state-action pair is visited an infinite number of times as the number of episodes from which the agent learns goes to infinity.

While the assumption of exploring starts is useful sometimes, with too large a state space it is impossible to visit all state-action pairs in this way.

4.8 On-Policy and Off-Policy Methods

There are two main types of methods for estimating the optimal policy using MC. These are on- and off-policy methods. On-policy methods will attempt to improve the policy that is used to make decisions, while off-policy methods will evaluate a policy different than that being used to generate the data. [5]

4.9 Greedy Policy

Greedy actions are ones in which an agent will always choose the action it believes will return the highest reward. [5] This means that the agent is not exploring new actions, and means that the agent will always choose a small immediate reward rather than an uncertain, but possibly larger, future reward. This can mean that the agent does not discover the optimal policy, as it never explores enough to find it, getting stuck with a policy it initially believes to be good.

4.10 Card Counting

Card counting is a process used by some blackjack players to gain additional information about the game apart from the cards in their hand. The Hi-Lo system [11] involves tallying numbers based on the cards that have already come up for the deck, before they have been shuffled in again. For each high card (face card, ten or ace) one is subtracted from the original count of zero, conversely for each card with value of six or lower, one is added to the count. By this method, with higher counts, more low cards have come up and therefore the higher cards are likely to come up next, which is advantageous to the player. This information can be used to make higher bets whilst the deck is favourable to the player.

4.11 Self Play

Self-play is where two agents learn to play the game by playing against each other. [2] This can result in different policies being learnt, introducing new ideas. This also means that neither player already has a fixed strategy, knowing only the rules, and so can learn from nothing, not even the other player.

5 Initial results

5.1 Introduction

Using the knowledge accrued in research thus far, we tackled the goal of using an autonomous algorithm to play blackjack. Below we discuss the ways we were able to approach this as well as how we knew how well we were doing.

5.2 Measuring Success

The efficacy of a reinforcement learning agent or algorithm is measured by its ability to gain rewards in its environment over time. This can be measured in two contrasting ways. One is the net number of wins, losses and draws after a certain number of iterations. This number is useful because it shows improvement over time. Since the knowledge of the agent starts at zero, its initial games are almost always the worst. This means a graph of win rate over time can show how the agent improves as it collects data and therefore knowledge about the environment. One downside to counting win rate results is that they are inherently limited by the environment in ways that the agent cannot control, for example in blackjack there is always a house edge that cannot be beaten without card counting or other amplifications of the simple state space. [5] Therefore when comparing win rates, it is useful to compare different agents or learning policies rather than comparing absolute results. The other way reward collection can be measured is to compare the state-action value function evaluated after many iterations to the true values for blackjack. This is a large table containing numbers for the expected reward after taking a certain action in a certain state. For example, it might say that with a player hand of 13, including a usable ace and a dealer card showing of 7, the result of hitting is 0.091 and the result of sticking is –0.497. This means that the agent will be able to pick the action with a higher expected outcome. This shows the final knowledge that an agent has gained about the environment and should eventually converge to the 'true' values for blackjack. Showing that an agent has reached these values shows that it is following the optimal policy for its environment and is at the limit for win rate. Learning algorithms that do reach the optimum policy can be compared by the speed at which they reach it, indicating their efficiency when it comes to evaluating policies. This observation benchmark will be useful for us to explore later in the project, especially when moving on to larger state spaces, as those algorithms that work faster will be far more useful then.

5.3 Turning blackjack into an algorithm

Blackjack is naturally an episodic finite MDP, with each game being an episode, and the rewards after each being +1 for winning, -1 for losing, and 0 for drawing. As each episode is only rewarded once, at the end when the result of the game is determined, all rewards within a game are 0, and there is no discounting. This means that, for blackjack, terminal rewards are also the returns.

The actions available in any state are a choice of two – hitting or sticking. The states in blackjack consist of the player's cards and the dealer's showing card. In our program, we will represent the state as three numbers, the total of the player's hand, the value of the dealer's visible card, and whether the player has a useable ace. To ensure there is no benefit in keeping track of the cards already dealt, the deck is assumed to be infinite. This will mean that an agent cannot learn to cheat.

We used an on-policy method, initialising our policy as sticking in every state. As the agent learned from experience, it could learn the expected return from taking each action in every possible state. Then, it would play the optimal action following what it had learnt. This would be its policy.

5.4 Base Environment

For any learning algorithm, an environment is required in which to test the agent. This is the program that provides the actual structure of the 'game' itself, taking actions as inputs from the player and returning the rewards that measure how well the game is being played. For blackjack, we initially coded a base environment that was playable by a human to gain an understanding of the game and test our casino skills. This was a useful exercise in programming which helped us develop a strategy for our first learning algorithms. Initially, we had based processes and calculations on a player hand represented by a list of the cards held, this was helpful for showing a human what cards they had but proved to be hard when it came to letting the computer play. In order to enact an autonomous policy an agent sees their 'state', this contains all the information that they know about the game which may influence their actions. After some trials, we decided that a simplified state showing the player's sum, dealer's sum and whether the player had a usable ace in hand would be easier to understand for agents. After rewriting the code for this state space, we were ready to evaluate some agents.

5.5 Agent-N

To test our base environment, we decided to create some simple algorithms whose policy is based entirely on the value of the player's hand. We named these Agent-N, with each individual algorithm have a name of the same form, with the number in place of N representing the lowest value of the player's hand on which the algorithm would stick.

Creating these algorithms meant that we were able to test the code for our base environment, ensuring that it could correctly pass the state and any other important pieces of information to an agent, as well as receive the correct action and play the game as intended. As our initial code for the environment used many global variables and did not pass variables such as the state between functions, we found it difficult to use this code to implement our agent, as it was hard to see where variables were being edited. We were creating our initial state by dealing cards as in a physical game using many functions to compete this task,

that were not working orthogonally. While this allowed us to show the game to a human player with more clarity, this information was irrelevant to an agent and resulted in a longer, less efficient code that made it more difficult for us to later change the state space, as each function relied on many others. This encouraged us to rewrite our code, so it was better planned, and each function had a separate purpose. This had many benefits, including a shorter, neater code, and functions that could be used for different purposes throughout our program as they were less tailored to one specific job.

After we had an ideal base environment, we were able to test our Agent-N algorithms and find the best policy using the simple idea of hitting on all hand totals below a value and sticking on the rest. As well as allowing us to test our environment, these algorithms also gave us a better understanding of what a good win rate would look like, and so what we should be aiming to beat. These algorithms, while simple, should be less effective than those we will create using reinforcement learning as they act purely on the total of the player's hand, not considering the dealer's hand, or whether the player has a usable ace, unlike the reinforcement learning agents we will code.

To test our Agent-N algorithms, we found the average percentage of wins for each algorithm. We did this by running each algorithm for 10,000 episodes, which each consisted of one game. The process of one game can be seen in Figure 5.1, which we ran many times to aggregate total rewards. As the algorithm does not learn, the percentage win rate should remain constant as the number of episodes increases and approach the true win rate for each algorithm. We chose 10,000 episodes as the number of episodes to test each algorithm on as it was high enough to give an accurate estimate of the average win rate for each algorithm, while also being quick to run which allowed us to run each algorithm multiple times. After doing this, we created a bar chart, Figure 5.2, which shows the average win rate for each Agent-N algorithm. It shows clearly that Agent-17 was the best agent, and as the value at which the agent sticks strays further from 17, the worse the algorithm performs. These results were very promising as this was what we had expected and gave us a win rate of 46.1% from Agent-17 to beat.

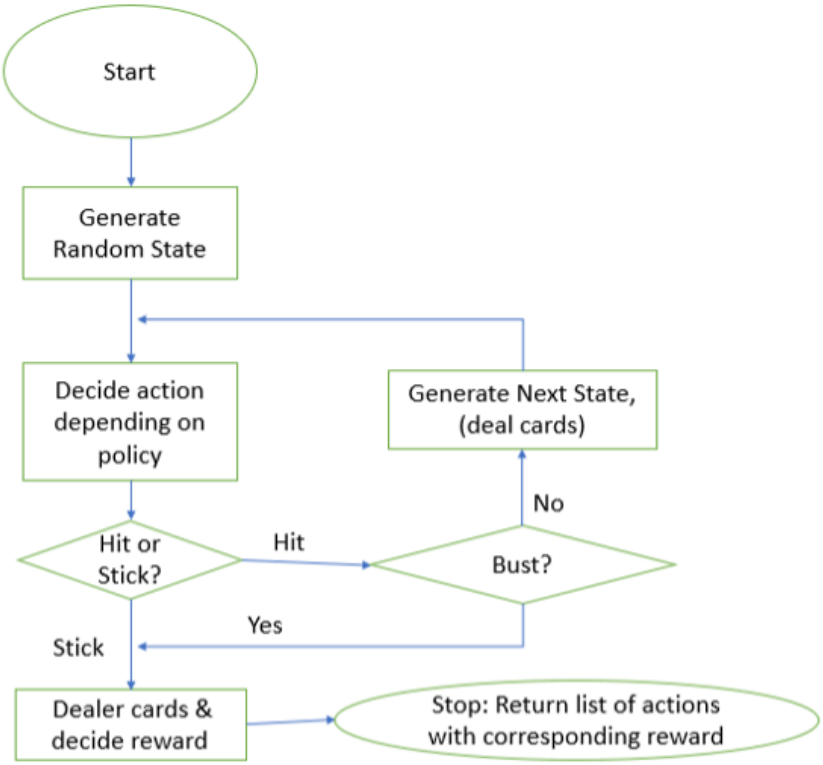

Figure 5.1: Flowchart for one episode

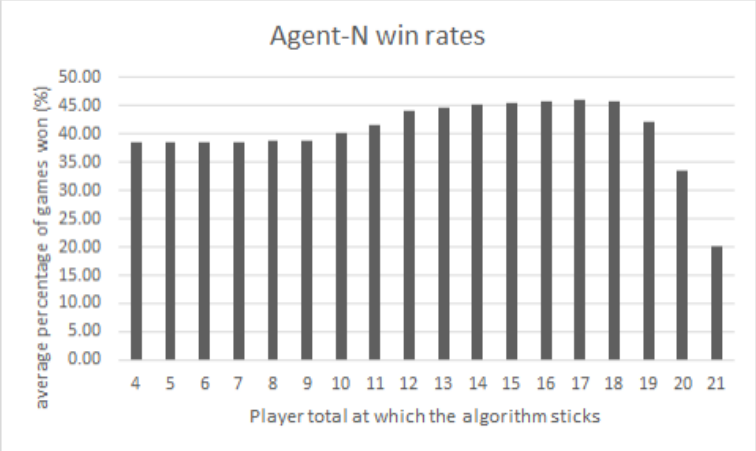

Figure 5.2: the win rates of our Agent-N algorithms

5.6 Random agent

Now we had found a highest win rate for a simple algorithm, we also wanted a lowest win rate that we should always hope to beat, even with a poor policy. We did this by choosing a random action every time, giving both hitting and sticking a probability of occurrence equal to 0.5.

With this, we found we achieved a win rate of around 38.5%. Therefore, none of our algorithms should ever be worse than 38.5% if they were learning a good algorithm, meaning we could check our programs worked by beating this figure.

5.7 Evaluation

The Agent-N algorithms proved very successful, showing that Agent-17 followed the best policy of this kind. This is what we had expected, as sticking on values of 17 and over is the algorithm that the dealer follows, and so should be the best policy for the dealer, to produce the highest house edge over the player.

Following this conclusion, we now should aim to beat this win rate as our reinforcement learning agent will use more information to calculate the best action in each state, and so with this added insight, should learn a better policy.

We can also use our best Agent-Ns to predict approximately what the optimal policy that we hope to find will be and use it to check that the policy created by our agent seems reasonable.

To improve our policy to reach a more successful game record, we will attempt to reach the optimal policy by playing the game and updating our estimates of the values of different actions in different states. The most valuable actions can be remembered in order to generate a better policy. In blackjack, we tried many different policies with sticking at different sums of the player hands and found average win rates, these were greatest when we were sticking at >= 17. The state parameters that we will consider in the future are the player sum from 12-21, as below these values the player cannot go bust, the dealer showing card and whether the player has a usable ace. With more information, the agent should be able to reach higher win rates.

5.8 Conclusion

Writing the environment and the Agent-N's has given us a much deeper understanding of how our reinforcement learning agent will have to operate, in areas such as the inputs and outputs that will be used. Now, we can begin to program our agent, as the next logical step to solving the problem of blackjack. Another useful outcome of this exercise is that we now have a benchmark of win rates to compare to in order to measure our agent.

6 Exploring Starts

6.1 Introduction

After having found a strong win rate to beat with our Agent-N algorithms, we were able to move onto writing our first reinforcement learning agent to learn to play blackjack. As discussed in our initial results section, blackjack is ideally suited to exploring starts due to its small state space, and so this was a good place to start attempting to solve blackjack.

6.2 Implementation of Exploring Starts

To implement exploring starts, each episode, our program generates a random initial state and action. The agent will then perform this action in this state. If the episode continues after this first action, then the agent will act greedily (following its learnt policy) in any further states until the episode ends. This ensures that the agent will gain experience in every possible state, leaving no gaps in its policy.

6.3 Learning

To record results, we created two large data structures to store our results. The first was a large dictionary, mapping every state-action pair to a long list of every return (win, loss, or draw) the agent had received following taking that action in that state. At the end of every episode, this would be updated, adding the return to all the state-action pairs that had occurred.

The second, and more important, data structure, was where we calculated our expected returns. Here, for every state-action pair, we averaged over every return stored with it in the dictionary and recorded this value. It is from this array that the policy can be calculated, as well as the value of each state. The policy of the agent is, in any state, the action with the greatest value. This will mean that the agent is performing the better action, according to its experience gained so far. This is the most important step of learning, as it helps the agent improve. The whole exploring starts process is shown in Figure 6.1, and the learning is shown in the loop where the reward list and policy is updated.

This will mean that, as the number of games that the agent plays tends towards infinity, the number of times the agent experiences each state tends to infinity. As this happens, the agent's calculated expected return following each action will converge on its true value, and the policy will reach the optimal policy.

6.4 Results

Having now written our exploring starts algorithm, we were ready to test our first reinforcement learning agent. To test it, we had to run the policy against the dealer, without the exploring starts first state-action pair. This action does not follow the agent's policy, and so

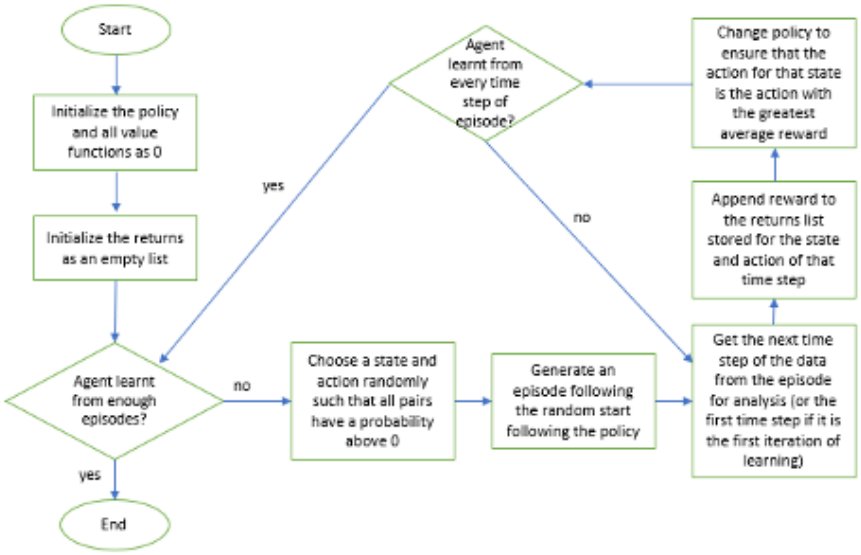

Figure 6.1: a flowchart showing the steps for teaching an agent using exploring starts

will not accurately show how well the agent is doing, especially as many games end after the first move.

After running our agent for over a million episodes, our agent had reached an almost optimal policy. It won around 48.7% of its episodes, beating the best agent-N algorithm, which had only won 46.1% of its episodes.

The policy that our agent calculated is shown below in Figure 6.2, with black areas showing where the agent decided to stick, and white areas where it found it optimal to hit. This policy is more nuanced and advanced than the agent-17, as for some values of the player hand, the policy will either hit or stick depending on the dealer card.

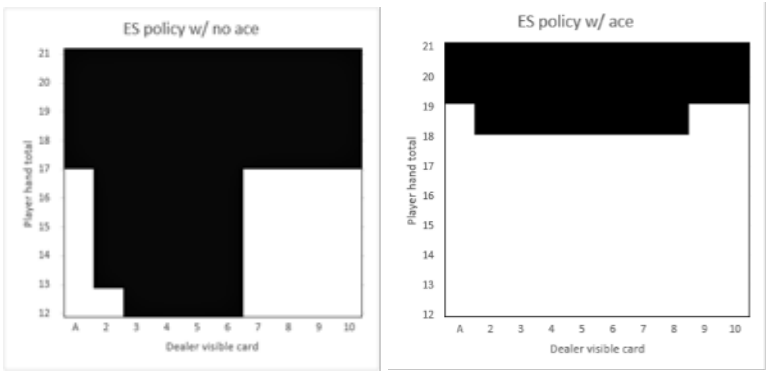

Figure 6.2: policies calculated by our agent

We were able to plot our state-value distribution for our policy using matplotlib, as shown in Figure 6.3. As our policy is the better action from each state, the value of any state following the policy is equal to the value of that better action. Therefore, we could use the data structure from before containing the state-action values to obtain are state-values.

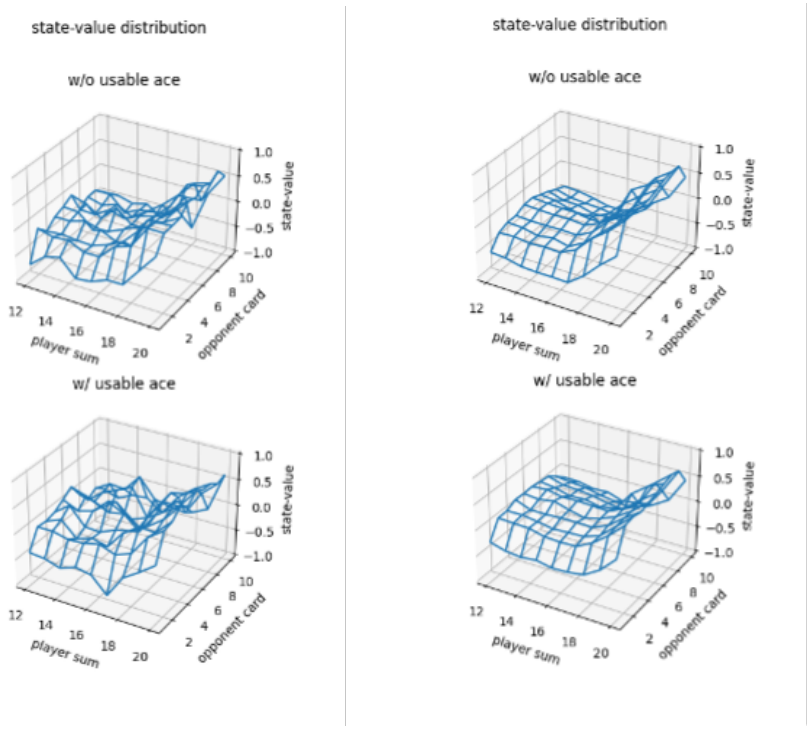

Figure 6.3: state-value distributions calculated after 30,000 games and 1,000,000 games

The above figure shows the state values after thirty thousand and one million episodes. The graphs flatten considerably as the number of games increases. This shows that the agent is approaching optimal values for the value of each state. As well as the results, we looked at how fast the agent approached its optimal values. Figure 6.4 shows the rate of learning. As can be seen by the trendline, the agent learnt most quickly at during the earlier episodes, and then slowly continued to approach the maximum number of episodes it won, as it approaches an optimal policy.

6.5 Extensions

After coding our agent, we were interested to see how our agent could be improved, by trying either other reinforcement learning techniques or by expanding the state space. A player of blackjack is always at a disadvantage against the house, as the player must go first and so could go bust, meaning the dealer wins immediately. We were interested to see whether this house edge could ever be overcome, or whether it would prove to be impossible, as casinos intend.

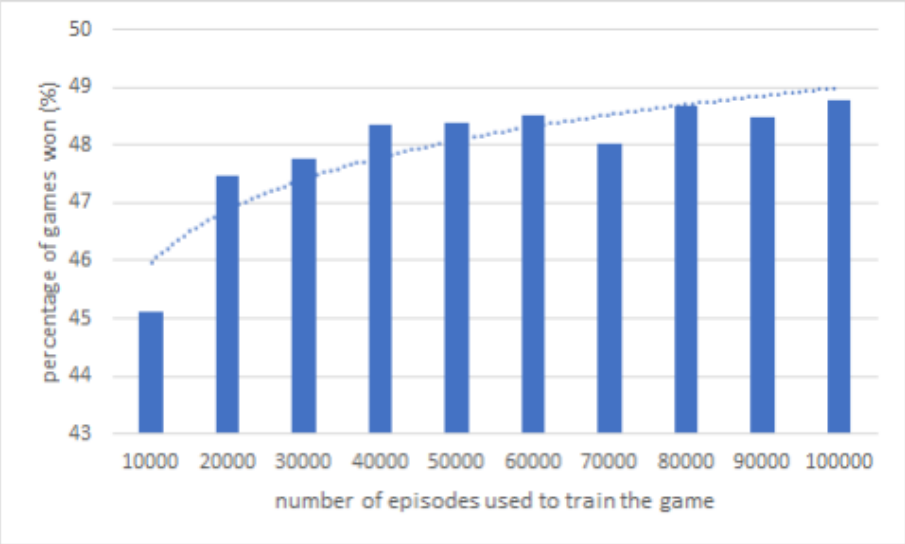

Figure 6.4: the win rate of our agent against the number of episodes it had experienced

6.6 Conclusion

Now we had successfully completed our project, as we had written a reinforcement learning agent, and solved the problem of blackjack. It performed better than the agent-N algorithm, as we hoped, but still was unable to beat the house edge.

We also had a much better understanding of both reinforcement learning and blackjack, and felt we were capable of improving and experimenting with the agent.

7 Greedy-only Algorithm

Another approach we investigated when approaching the blackjack problem was going for a greedy-only algorithm. One of the disadvantages of this approach, however, is it prioritises exploitation against exploration and therefore, it would have a higher immediate reward but a potentially lower final reward. In an attempt to combat this, we started from the end state (when the player decided to stick) and worked our way back. Metaphorically, imagine a maze, some may find the maze easier to complete if they start from the end and work back towards the start. Similarly, we evaluate the value of the final states, where the winner has been decided, and branch our way in, giving values to only the highest of states, essentially just a reversed binary tree [See Figure 7.1]. This way, it enabled having higher longer-term rewards.

Comparing this approach with the Monte Carlo method, the win rates were lower, however was compensated by a higher draw rate. Overall, this does make this method obsolete, as preferably, you would want to win a higher number of games rather than drawing them. One of the reasons for this may be due to the fact that in the first states, the rewards are unpredictable and therefore, they may not be as rewarding overall. Also, the greedy policy would only continue to branch out from the highest rewarding states, ignoring those with not

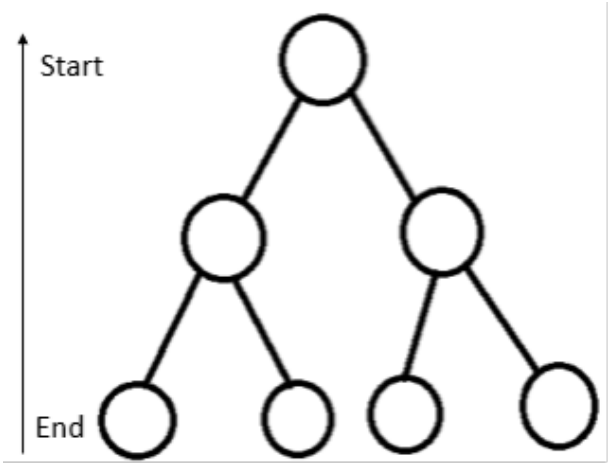

Figure 7.1 [Diagram of a tree of results where the probabilities of the final states are found first]

as high of a reward. Therefore, it may result in some options being overlooked only due to the fact the final state is not as rewarding. So, in the long term in terms of games, it would priorities drawing, or not losing points instead of gaining them. On the other hand, had there been more training plays, more branches would have been explored and therefore, a more optimal policy may have been found but overall, we found the greedy-policy algorithm less efficient than the exploring-starts algorithm.

8 Card Counting and State Expansion

8.1 Motivation & Expanding State Spaces

Through our endeavours with our exploring start algorithm, it became clear that the maximum possible win rate that we would achieve was inherently limited by the game of blackjack. As casinos are businesses that must make money to survive, the odds of blackjack are in favour of the house when playing normally. Playing normally, in this case, means playing how the casino expects you to play: betting first with no information, and then using available information of the cards in the player's and dealer's hands to inform actions of hitting or sticking. This design means that even when following the optimum policy, our agent would lose most of the time. An ideal reinforcement learning algorithm will find the optimum policy for achieving maximum rewards in an environment given the information it has access to and the options it has access to, meaning one way of increasing performance is by increasing information accessible to the agent. In practical terms, this meant we had to expand the 'state' visible to our algorithm when making each decision.

8.2 Card Counting and Finite Decks

One method famously used to gain an edge over the house in blackjack by gathering more information is the process of card counting (as explained in key concepts). In short, it involves numerically keeping track of which cards have already come up in the deck to track whether cards that favour the player are more likely to come up in the near future. In the Hi-Lo [11] card counting system, a high count is good for the player, and takes an integer form that can range from negative to positive, with higher values showing increased chances of good events. In order to implement this in our code, we first had to overhaul it so that the deck was no longer 'infinite', this involved having a data structure representing the physical deck, from which numbers were gradually taken, representing the cards being taking out. This was critically different from before because the finite deck means the probabilities of drawing different cards changes drastically depending on which are left in the deck, changing the balance of the game. When blackjack card counters play, they use the count to influence only their bet. This means they will bet big when the count is favourable, and then play the round with normal strategy, hoping that the high count means they are more likely to win the round. This is sensible for a human being because memorising different ideal actions to follow for all the different scenarios and all the different counts is impossible, however reinforcement learning gave us the tools to investigate whether there is a better way to play, where not only the bet size but also the strategy changes with the count.

8.3 Results and Observations

Once the deck had been made finite, the next step was to expand the state space that the algorithm would observe and understand. This was done by adding a fourth parameter (alongside player total, dealer card and useable ace) which was the count. Initially, we began with an integer range from –15 to 15. We followed the same exploring starts algorithm detailed above by creating the first random state of each episode as if the player had just sat down in the middle of an ongoing blackjack session, part-way through a finite deck where they knew the count leading up to the episode they were about to play. When we tested this however the results were erratic and did not show that more information had in any way improved win rates, in fact, they had worsened (to around 38.5%). Unbeknownst to us, we had just encountered Bellman's famous Curse of Dimensionality [12] of reinforcement learning. The simplest way to explain this is that expanding the number of variables looked at in the state (here we did so by a factor of 31) expands the number of boxes in the 'lookup table' (state-action value function) telling the program what to do at each step. For the program to be able to accurately distinguish between actions, it has to have tried each action in a certain state many times to know how good an action really is in that state, meaning that if the number of 'boxes' in the table is made larger (by discriminating different states using count), the program must repeat far more random iterations until it converges with the true policy and state-value function. With the initial card counting method, the problem

was exacerbated by the fact that not all states were distributed with equal likelihood, for example, an extremely high count such as +13 can only be reached when 13 low cards come in a row which is exceedingly unlikely with a randomly shuffled deck. This meant that fringe states with very high or very low counts had very few visits when the program was run with few iterations, meaning the data it had on different actions in those states was scarce and unreliable at best, and non-existent at worst. The number of visits to states in respective counts is shown in Fig. 8.1

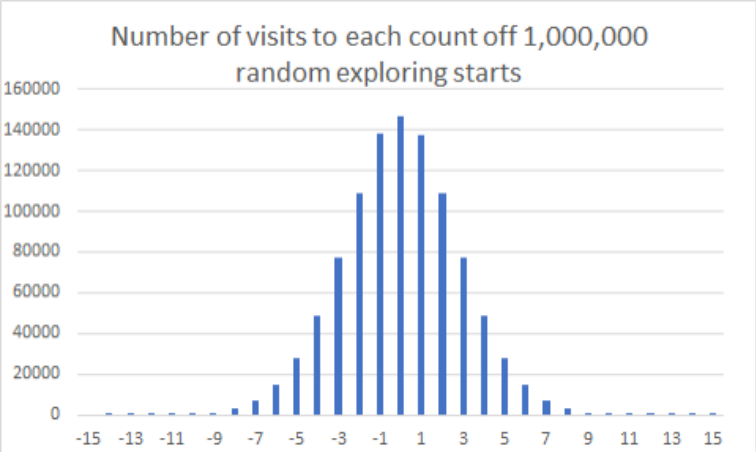

Figure 8.1 The number of visits to states in respective counts

This was problematic, particularly in the high count states where the program should be more likely to win, because even though the high count is slightly more favourable to the player, this assumes perfect strategy and a lack of data meant the program was playing erratically, destroying hopes of ever beating the house edge by winning in these states.

There are three obvious ways to counteract a lack of data obtained through random Monte Carlo iteration, the first is to simply play more times until the policy converges with the optimum anyway. As a group however, we concluded that this solution was a useful stopgap measure to increase the performance of an algorithm by a few percentage points, but not enough to immediately solve the problem. This is because a brute force approach such as this one is heavily resource intensive, and increasingly impractical as the scale increases. For example, for a state space of 31 times the size of the original, a minimum of 31 times the runtime is required to amass the same average amount of data per state action pair as before. Even more unfortunately, this doesn't fix the problem of an uneven distribution of the probabilities of reaching different states: if a state previously visited very few times is visited 31 times more this still may not be enough to obtain sufficient data for effective play.

Another way of dealing with the Curse of Dimensionality, which directly addresses the uneven state probability distribution is deterministically visiting all of the states in order, one by one, repeating until each state has been visited enough times. Although this is a common technique in reinforcement learning, for the game of blackjack this would have been impossible, because the exact status of the deck is required by the environment and cannot be generated purely from a state. This is because in this Markov Decision Process the exact model of probabilities of outcomes in the environment is not known to the agent. (Incidentally, this is the same problem that forced us to use Monte Carlo methods to estimate these probabilities).

The last way, to overcome the Curse of Dimensionality is 'the art of state aggregation' as described by Uther et al [13]. This involves reducing the dimensions of a state to avoid the complications of a larger state space, without losing the new information of the larger state space. In short, environment scenarios that are similar enough in their likelihood of resulting in a successful reward for a given action can be grouped and viewed as the same state by the program because they do not merit different actions. Therefore, after some analysis to see how often different counts were visited, it became clear that card counts of much higher than five occurred so rarely that they were the ones with significant data lacking, causing erratic actions. Our new state-space comprised of the same three initial variables, and a new count variable, which was an integer from -5 to +5, where the top and bottom counts included all higher or lower counts.

This vastly reduced the size of the state space to explore and meant that the program was able to visit states more consistently as shown in Fig 8.2

Intriguingly, although states were now being visited more consistently win rates still stayed lower than with previous algorithms (43.2% at optimum policy). Interestingly the

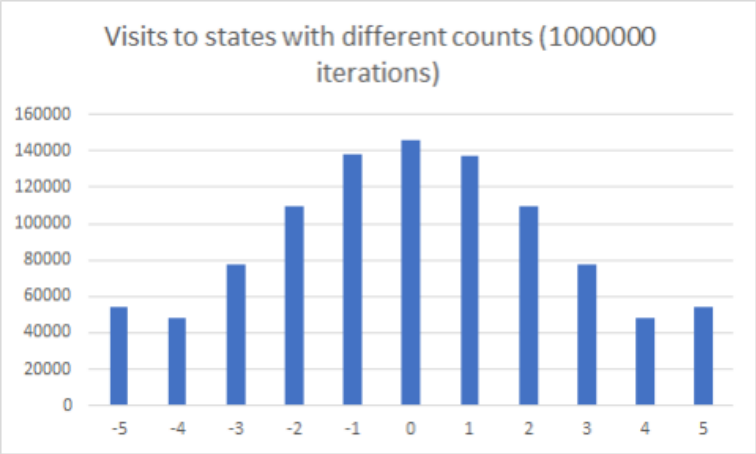

Figure 8.2 The number of visits to states in respective counts in reduced state space size

distribution of win rates by count was no higher with higher counts as expected (Shown in Fig 8.3)

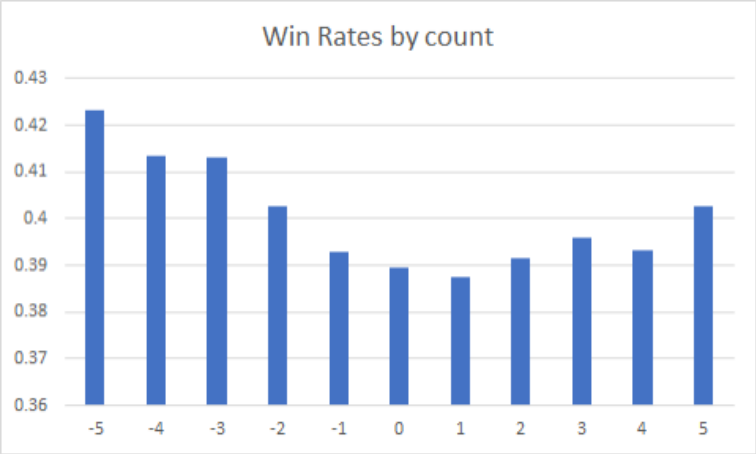

Figure 8.3 The win-rate by count with increased state visits

One hypothesis about the substandard win rate is that the truly desirable very high count states were incorrectly aggregated with all other states above five, meaning actions were taken incorrectly. This is unlikely however because the states are visited so rarely that their effect on the average should be small. Another hypothesis is that, because different policies are being played at each count, the lower natural probability of winning by chance in lower count states meant that accurate play was more likely to be rewarded, whereas in higher count states, coincidental wins were more likely to occur, meaning positive rewards were not always tied to optimum actions. If this is the case, it indicates that with a similar number of iterations to other algorithms, the optimum policy has still not been reached and there is a simple solution of applying more processing time to the running of the program as Monte Carlo exploring starts will always reach the optimal policy with enough iterations.

8.4 Evaluation and Future Ideas

As we further investigated expanding state spaces the limitations of reinforcement learning became very apparent, answering the question of why computers do not yet rule the world. As more dimensions and possibilities of data are added, simple algorithms become very quickly overwhelmed by the exponentially growing processing time and diminishing returns on rewards. In addition, difficulties are caused for programmers, when data can no longer be easily visually displayed, meaning diagnostic checks of whether the program is working correctly become much harder. This seems to point towards the role of reinforcement learning in the future being a tool to aid humans with severe limits, best used in tandem with known human knowledge about a subject. For example, a RL algorithm could be used with the Kelly Criterion (mathematics for calculating wager size depending on success chance) [14] to calculate optimum bet sizes at casinos in future investigations.

9 Self-play

9.1 Introduction

Having written the exploring starts code, it was easy to manipulate what we had written to implement self-play. We wanted to investigate how effective the policies that the new agents would learn would be against a dealer, as well as how these policies might differ from the optimal policy found by our exploring starts code.

9.2 Implementation

To implement self-play, we edited our exploring starts code to allow for two agents to play each other. One necessary change was to alter the way states were calculated. Previously, the state had purely been the three numbers that the agent would see – the player total, the dealer card, and whether the player had a useable ace. The individual cards did not need to be known, and any further cards for the dealer were relevant only after the player had finished their turn, and so did not affect the state.

However, when implementing self-play, we wanted the experience of both players to mimic the experience of the player against a dealer. Therefore, both would be given a state containing their total, one of the other's cards, and whether they had a useable ace. This meant that the environment had to know all cards in play, to be able to show either the total of a hand or just one card in it, depending on which player needed the information.

The other change was how the experience from the episodes would be used for an agent to learn. First, we set it so that the two agents would learn different policies from their own experiences. This meant we could compare policies and their respective win rates. After, we looked at how both agents could learn to the same policy, combining all experience into one policy.

9.3 Two policies

Within this, we wanted to look at how the order in which the agents played would affect the policies that they learnt, and, if they learnt different policies, how the policies would compare when playing against a dealer.

First, we set it so that one agent always played second, and the other always played first. This meant the first agent had a similar situation to that of the player, and the second the dealer. Therefore, during training, it seemed very likely that the second player would often beat the first, as they would have an edge like the dealers. However, we were uncertain as to how this could affect the policies, and which would eventually be the best against the dealer.

As we expected, the second player performed better during training, winning around 48.4% of games and losing 46.3%. Both players performed quite badly against a dealer, performing worse than our benchmark of a 'good' grade from agent-17. This was to be expected as neither were trained to play against a dealer, and as both had started from completely random choices, their initial state-action values were likely to be very inaccurate.

The second agent always performed marginally better, achieving a win rate of around 43.7%, while the first achieved around 43.3%. The policies can be seen below in Figure 9.1, compared to each other and also to the optimal policy found by the exploring starts agent. While they have similar shapes, they are still quite different. This explains partially why the second agent would perform better – the policy that it learns is closer to that of the exploring starts than the policy learnt by the first player. We thought it interesting that the second player performed better, even though the first was learning in a more similar environment than against the dealer.

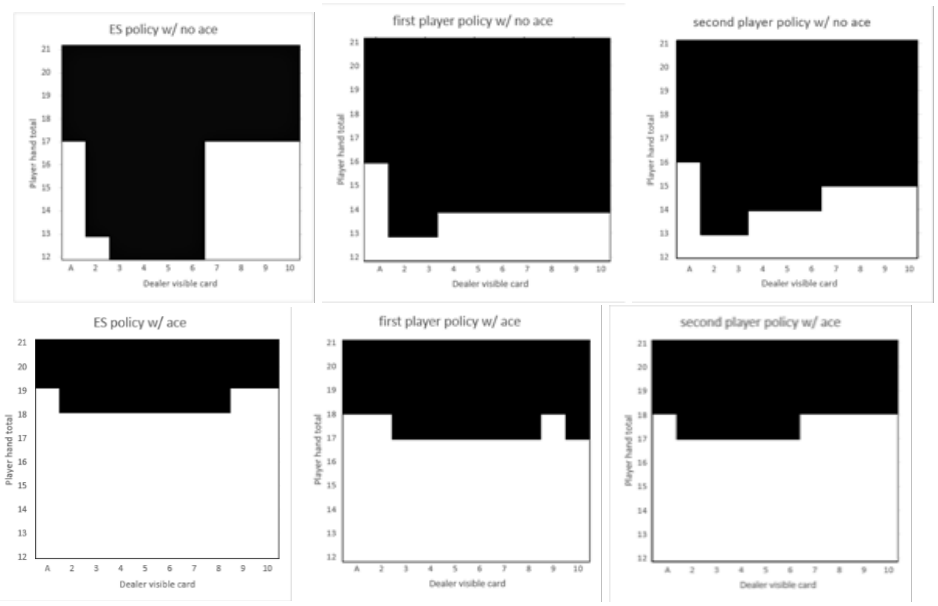

Figure 9.1: the policies learnt by exploring starts and self-play

Now, we wanted to explore what would happen if we had the two agents swapping which went first every episode. We were interested to see whether this resulted in different policies, or they would now reach the same policy, as they were in a much more similar situation.

The policies for the two agents are shown below in Figure 9.2. Here, the policies are very similar, meaning that the agents learnt the same policy when playing against each other now they were in the same situation, rather than different strategies.

The win rates for these policies also came out similar, around 43.1%. This is similar but slightly lower to the previous self-play win rates, further confirming that self-play will not achieve the optimal policy. This is to be expected, as the agents are training in a slightly different scenario to that they are being tested in, whereas the exploring starts agent is trained in exactly the scenario it is being tested in.

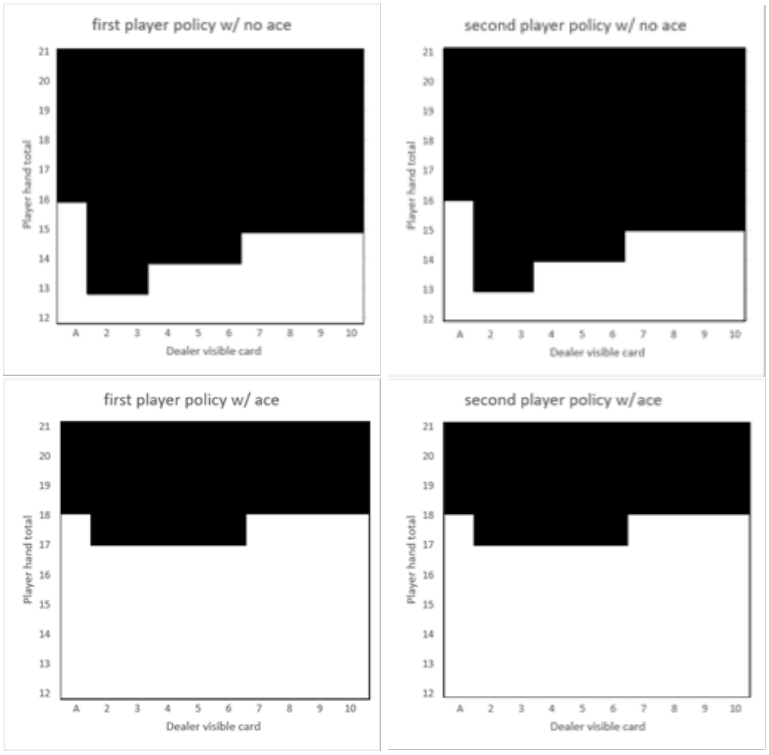

Figure 9.2: comparison of the two policies for alternating self-play

9.4 One policy

The other way to implement self-play is for both agents to learn to and from the same policy. The policy that this method reached after the same number of episodes, 3,000,000, was not as 'neat' looking as others, as shown in Figure 9.3. This could imply either that it had not reached its optimal policy, or that this method results in a policy like this. To work out which of these was likely to be the case, we ran the algorithm multiple times to compare policies. However, no runs would give us 'neater' policies, and so to truly test this, many more runs would be needed, something we could not do with limited computing power.

Despite this ‘messier’ appearance, the win rate for this came out as similar to the other two self-play algorithms, or around 43.7%.

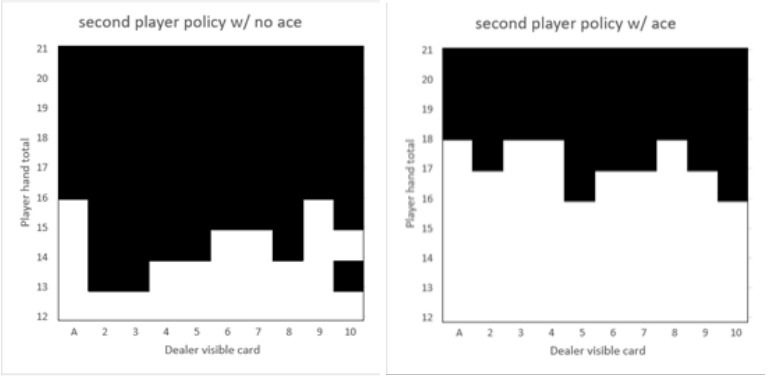

Figure 9.3: one of many similar policies given by the one policy self-play

9.5 Conclusion

Overall, while our self-play algorithms did not reach the optimal policy, it was very interesting to see how two agents could still reach a policy similar to the optimal policy when both learning to play at the same time. This meant that they would both evolve together, continually improving to meet the other’s improvements, and this exercise allowed us to learn a lot about how useful self-play could be in a different situation, as these slightly different techniques could allow many policies to be found and tested.

This extension has raised many questions and we think that it would be interesting to continue this research in self-play with blackjack and test more variations on order and policy number could achieve a better win rate. It would also be interesting to see whether running the training for even more episodes than the 3,000,000 we had been using would result in the agents getting better, but this is not something that we had the time to investigate.

10 Conclusion

In this project, we solved blackjack using several different methods of reinforcement learning. We researched and implemented many algorithms, including several Monte Carlo techniques. This allowed us to compare and analyse various reinforcement learning methods, both in general and with the problem of blackjack.

We initially developed our Agent-N algorithm, and found the win rate for random decisions, giving us benchmarks to compare our agents against. These also taught us a lot about the best ways to structure a program, writing it in such a way that it would be easy to implement an agent to decide the moves.

We then wrote our first reinforcement learning agent using exploring starts, a Monte Carlo method. This method was ideal for our problem, due to our small state space, and the fact that it would not be possible to model our environment due to the random cards being

dealt. It allowed us to find the optimal policy for our problem, giving us the best solution to the problem without cheating. We achieved a win rate of 48.7% with this, impressively close to 50%. It is impossible to breach 50%, as the dealer will always have an edge, so getting within 1.5% of this value was a great result.

We then tried an algorithm implementing a much greedier policy. While this technique can be very good for large state spaces where it is impossible to visit all states, as well as environments which can be modelled, but we found it less ideal for blackjack as neither of these criteria are true here. We were unable to find the optimal policy, as this technique can be less consistent, but using more training episodes could have improved our results.

After these two algorithms, we tried two other extensions. The first of these was training two algorithms simultaneously using self-play. We tried various versions of this, changing both the order of the agents, and the number of policies that they created. While some of these were very close to an optimal policy, none managed it. This is likely to be because the agents were learning in a slightly different environment to that they were being tested in, as they were playing another agent rather than the dealer.

Finally, we attempted to implement card-counting. As this would increase the amount of information the agent would have, it would allow the agent to achieve a better policy, potentially even breaking through the 50% win rate. However, we were unable to finish this, as the state space grew too quickly for us to implement it in the best way with exploring starts.

Overall, we successfully solved blackjack, using various techniques. We managed to find the optimal policy for blackjack and were able to compare various reinforcement learning agent. In the future, it would be interesting to look further into whether the optimal policy can be found with the greedy policy algorithm, or with self-play. Also, finishing card counting, with potentially a different reinforcement learning technique could even result in a positive win rate.

Finally, as a group, we are very proud of the algorithms developed and the results we have achieved; exploring reinforcement learning has been a great experience, as well as a great opportunity to collaborate on a large coding project. We hope our work on blackjack will aid the advancement of reinforcement learning whilst providing a medium to teach others the foundations of a fascinating subject.

References

[1] Murali A. The importance of Reinforcement Learning. Perficient 2019; Available from: `https://blogs.perficient.com/2019/10/08/reinforcement-learning-in-ml/` [Accessed 13th December 2020].

[2] Silver D, Hubert T, Schrittweiser J, Hassabis D. AlphaZero: shedding new light on chess, shogi and go DeepMind 2018; Available from: `https://deepmind.com/blog/`

article/alphazero-shedding-new-light-grand-games-chess-shogi-and-go [Accessed 13th December 2020].

[3] Senior A, Hassabis D et al. AlphaFold: a solution to a 50-year-old grand challenge in biology. DeepMind 2020; Available from: https://deepmind.com/blog/article/alphafold-a-solution-to-a-50-year-old-grand-challenge-in-biology [Accessed 15th December 2020].

[4] Cameron E, Andrews J, Gillham J. The economic impact of artificial intelligence on the UK economy. PricewaterhouseCoopers 2017; Available from: https://www.pwc.co.uk/economic-services/assets/ai-uk-report-v2.pdf [Accessed 13th December 2020].

[5] Richard S, Andrew B. Reinforcement Learning. 2nd ed. London: The MIT Press; 2018; Available from http://www.incompleteideas.net/book/RLbook2020.pdf [Accessed on 29th November 2020].

[6] Ldn-Post. Statistics on online gambling in the UK. London Post 2020; Available from: https://london-post.co.uk/statistics-on-online-gambling-in-the-uk/ [Accessed 13th December 2020].

[7] Dittert S. Monte Carlo Methods in Reinforcement Learning – Part 1 on-policy Methods. Medium. 2020; Available from: https://sebastian-dittert.medium.com/ [Accessed 13th December 2020].

[8] Torres J., The Bellman Equation- Q function and V function explained, Towards Data Science, 2020; Available from: https://towardsdatascience.com/the-bellman-equation-59258a0d3fa7 [Accessed 13th December 2020].

[9] Martinez G. Dynamic programming (DP), Monte Carlo (MC) and Temporal difference (TD) to solve the grid world state-value function: Reinforcement learning (RL) 101 with Python. Towards Data Science; 2018; Available from: https://towardsdatascience.com/reinforcement-learning-rl-101-with-python-e1aa0d37d43b [Accessed: 23 November 2020].

[10] Lazaric A, Restelli M, Bonarini A. Reinforcement Learning in Continuous Action Spaces through Sequential Monte Carlo Methods. Advances in Neural Information Processing Systems. 2007; Available from: https://papers.nips.cc/paper/2007/file/0f840be9b8db4d3fbd5ba2ce59211f55-Paper.pdf [Accessed 29th November 2020].

[11] Zutis K, Hoey J. Who's counting? Real-time blackjack monitoring for card counting detection. In Fritz M, Schiele B, Piater JH, editors, Computer Vision Systems: 7th International Conference on Computer Vision Systems, ICVS 2009 Liège, Belgium, October

13-15, 2009. Proceedings. Berlin: Springer. 2009. p. 354-363. (Lecture notes in computer science); Available from: https://doi.org/10.1007/978-3-642-04667-4_36 [Accessed 23rd June 2021].

[12] Bellman R. On the theory of dynamic programming. Proceedings of the National Academy of Sciences of the United States of America. 1952 Aug;38(8):716; Available from: https://www.semanticscholar.org/paper/On-the-Theory-of-Dynamic-Programming.-Bellman/dc9047917d1ceb3805d954c73899ddd2d40dd5eb [Accessed 23rd June 2021].

[13] Uther WT, Veloso MM. Tree based discretization for continuous state space reinforcement learning. Aaai/iaai. 1998 Jul 1;98:769-74; Available from: https://www.aaai.org/Papers/AAAI/1998/AAAI98-109.pdf [Accessed 23rd June 2021].

[14] Thorp EO. The Kelly criterion in blackjack sports betting, and the stock market. The Kelly capital growth investment criterion: theory and practice 2011 (pp. 789-832); Available from: http://www.eecs.harvard.edu/cs286r/courses/fall12/papers/Thorpe_KellyCriterion2007.pdf [Accessed 23rd June 2021]